

How AD Can Help Solve Differential-Algebraic Equations

John D Pryce^{a*} and Nedialko S Nedialkov^b and Guangning Tan^c and Xiao Li^d

^aCardiff University School of Mathematics, Senghennydd Rd, Cardiff, CF24 4AG, Wales;

^bMcMaster University Department of Computing & Software, Hamilton, L8S 4K1, Canada.

^cMassachusetts Institute of Technology, Process Systems Engineering Laboratory, Boston, USA.

^dMcMaster University School of Computational Science and Engineering, Hamilton, L8S 4K1, Canada.

A characteristic feature of differential-algebraic equations is that one needs to find derivatives of some of their equations with respect to time, as part of so called index reduction or regularisation, to prepare them for numerical solution. This is often done with the help of a computer algebra system. We show in two significant cases that it can be done efficiently by pure algorithmic differentiation. The first is the Dummy Derivatives method; here we give a mainly theoretical description, with tutorial examples. The second is the solution of a mechanical system directly from its Lagrangian formulation. Here we outline the theory and show several non-trivial examples of using the “Lagrangian facility” of the Nedialkov-Pryce initial-value solver DAETS, namely: a spring-mass-multipendulum system; a prescribed-trajectory control problem; and long-time integration of a model of the outer planets of the solar system, taken from the DETEST testing package for ODE solvers.

1. Introduction

1.1 DAE formulation and basic ideas

In industrial engineering, the modeling of systems to simulate their time evolution is increasingly done by methods that lead to a differential-algebraic equation (DAE) system as the underlying mathematical form. Such DAEs often come from equation-based modeling (EBM), which describes system components by the basic physical laws they obey and supports “multiphysics” models that combine several scientific disciplines, as for instance mechanical, electrical, chemical, and thermodynamic behaviour in a car engine.

Facilities created to support EBM include gPROMS, which is both a language and a graphical modeling environment (GME) built on it; the Modelica language and GMEs such as OpenModelica, Dymola and MapleSim that are built on it. Simulink, built on MATLAB, is a GME of similar scope but less in tune with the general DAE concept.

A DAE is just a set of n equations connecting a vector $\mathbf{x} = \mathbf{x}(t)$ of n state variables x_1, \dots, x_n and some derivatives of them with respect to time t . One can always reduce it to a first order form $\mathbf{F}(t, \mathbf{x}, \dot{\mathbf{x}}) = 0$ —as accepted by the DASSL solver and its relatives [1, 6]—in the same way as one does for an ODE system. Here $\dot{\mathbf{x}}$ means $d\mathbf{x}/dt$. However we use a more flexible form allowing arbitrary higher derivatives:

$$f_i(t, \text{the } x_j \text{ and derivatives of them}) = 0, \quad i = 1, \dots, n. \quad (1)$$

This often lets one formulate problems to our DAETS initial-value code [9, 10] more concisely, e.g. Lagrange’s equations for a mechanical system with n_q coordinates and n_c constraints need $n_q + n_c$ variables, compared to $2n_q + n_c$ in first-order form.

1.2 Aim

In general, differentiating some of the DAE’s equations $f_i = 0$ with respect to t is an essential step in solving a DAE. This article is about two significant and rather different uses of this. The first is the widely used *dummy derivatives* method of Mattsson & Söderlind [7] that prepares a higher-index DAE for numerical solution by a classical index-1 DAE code, or by an explicit ODE code such as a Runge–Kutta method.

The second is the task of solving a, possibly constrained, mechanical system directly from a Lagrangian formulation. Conceptually it has a “two-phase” aspect. The motion of the system is defined by a *Lagrangian function* $L(q, \dot{q})$ where $q = (q_1, \dots, q_n)$ is a vector of generalised coordinates, together with m constraints $C(q) = (C_1(q), \dots, C_m(q)) = 0$. To set up (phase 1) the equations of motion from L and C involves partial differentiation $\partial/\partial q$ and $\partial/\partial \dot{q}$ applied to L and C , as well as straight differentiation d/dt . When $m > 0$ the result is a DAE, which must (phase 2) be readied for numerical solution.

Each of these use cases at first sight seems to need symbolic differentiation, e.g. in a computer algebra system. We show pure AD suffices in either case. In the second case, which we devote more space to, AD can even do the two “phases” simultaneously, and gives a simple and elegant user interface and an efficient numerical solution process.

1.3 Structural analysis

In an ODE $\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x})$, causality is obvious: in differential language, it explicitly specifies the state $\mathbf{x} + d\mathbf{x}$ at the next instant $t + dt$ to be $\mathbf{x} + \mathbf{f}(t, \mathbf{x})dt$.

In a DAE, causality is not obvious. For instance, these size 2 DAEs are quite different, where $u(t)$ is a given driving function:

$$\begin{aligned} x_1 - u(t) &= 0, & x_1 - \dot{x}_2 &= 0, & (2) \\ \text{and } x_2 - u(t) &= 0, & x_1 - \dot{x}_2 &= 0 & (3) \end{aligned}$$

To solve (2), make \dot{x}_2 the subject of its second equation (x_1 causes x_2) and integrate the result; it is really an ODE, with one degree of freedom. To solve (3), make x_1 the subject of its second equation (x_2 causes x_1) and differentiate. DAE (3) has no degrees of freedom—it has the unique solution $x_1 = \dot{u}(t)$, $x_2 = u(t)$ and does not look like an ODE at all; such behaviour is common in control problems.

A solvable DAE has a chain of causality that must be found in order to prepare for numerical solution. Knowing which equations $f_i = 0$ to differentiate, and how often, is crucial to finding this causal chain. When correctly done, the original DAE augmented by the differentiated equations can be solved to produce an ODE in some (possibly not all) of the original variables—the *ODE part*. Once this ODE is solved, the remaining variables forming the *algebraic part* can be found by algebraic manipulations combined with differentiations.

Let c_i be the number of differentiations of equation i needed by the “most economical” way of doing this. For reasons to do with the Taylor series method used by DAETS we call them the *equation-offsets*.

For instance the equations of (2) do not need differentiating: $(c_1, c_2) = (0, 0)$. We solve to produce the ODE part $\dot{x}_2 = u(t)$ in just x_2 . By contrast, (3) has $(c_1, c_2) = (1, 0)$ meaning the first equation must be differentiated, after which we solve to get $x_1 = \dot{u}(t)$, $x_2 = u(t)$. The ODE part is empty.

In the DAE (2), it happens we can solve for the algebraic variable x_1 to get $x_1 = u(t)$, independently of solving the ODE, but this need not be so: if we change it to

$$x_1 - x_2 - u(t) = 0, \quad x_1 - \dot{x}_2 = 0, \quad (4)$$

then the ODE part, namely $\dot{x}_2 - x_2 - u(t) = 0$, must be solved before we know x_1 .

Unlike a well-behaved ODE $\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x})$, which has a solution path through each point of the region R of (t, \mathbf{x}) space where it is defined, the union of a typical DAE's solution paths is a proper subset of R , the *consistent manifold* \mathcal{M} or set of *consistent points*. The dimension of its intersection with any time $t = t_0$ is DOF, the number of *degrees of freedom*, equivalently the size of its ODE part (here assumed independent of t_0).

The *index* of a DAE used in this paper is simply

$$\nu = \max_i c_i. \quad (5)$$

The classic *differentiation index* ν_d assigns index 1 and 2 to these DAEs respectively. In summary for the examples above

DAE	ODE part	DOF	algebraic part	offsets	ν	ν_d
eqns (2)	x_2	1	x_1 (found independently of ODE part)	$(0, 0)$	0	1
eqns (3)	empty	0	x_1, x_2	$(1, 0)$	1	2
eqns (4)	x_2	1	x_1 (found using x_2 in ODE part)	$(0, 0)$	0	1

The *structural analysis* (SA) approach aims to derive a DAE's causal chain by studying its sparsity, namely what derivatives of variables occur in what equations. The method is: seek a number c_i of times to differentiate the i th equation that gives a *structurally nonsingular* (SNS) set of equations for the resulting highest, d_j th, derivatives of the x_j —then $\mathbf{c} = (c_1, \dots, c_n)$, $\mathbf{d} = (d_1, \dots, d_n)$ are the vectors of equation-offsets and corresponding variable-offsets. SNS means one can make a matching of variables to equations—equivalently a *transversal*, a set T of n positions (i, j) in an $n \times n$ matrix with just one in each row and in each column—such that derivative $x_j^{(d_j)}$ occurs in the differentiated equation $f_i^{(c_i)} = 0$ for each $(i, j) \in T$. There exist unique elementwise smallest non-negative \mathbf{c}, \mathbf{d} , the *canonical offsets*, which we assume chosen henceforth. They define the “most economical” differentiations mentioned above.

An *SA-friendly* DAE by definition is one for which these equations are actually (not just structurally) nonsingular at some consistent point, that is, the $n \times n$ *system Jacobian*

$$\mathbf{J} = \left(\partial f_i^{(c_i)} / \partial x_j^{(d_j)} \right)_{i,j=1,\dots,n}. \quad (6)$$

is nonsingular there. Assuming suitable smoothness of the f_i , a unique solution then exists locally through this point, and through any nearby consistent points.

Experience shows most DAEs in practice are SA-friendly. This fact underlies the wide use of the dummy derivatives method, which uses the results of SA and succeeds if and only if the DAE is SA-friendly. The SA can be done by the graph-based Pantelides

method [11], or the Pryce Σ -method [12] based on the *signature matrix* $\Sigma = (\sigma_{ij})$, where

$$\sigma_{ij} = \begin{cases} \text{order of highest derivative of } x_j \text{ in } f_i & \text{if } x_j \text{ occurs in } f_i, \\ -\infty & \text{if not.} \end{cases} \quad (7)$$

The methods are equivalent except that the latter handles higher-order DAEs without reduction to first order, while the former as described in [11] does not.

The DAE (of classical index 3) derived from a constrained Lagrangian of a mechanical system as in Section 3, is always SA-friendly when posed as an initial value problem. Posed otherwise, e.g. as a prescribed-trajectory control problem, it need not be. The occurrence of non-SA-friendly but solvable DAEs in applications is studied in [14, 16]. For systematic ways of converting such a DAE to an equivalent SA-friendly one see [18].

2. Dummy derivatives

2.1 The DDs construction

Many numerical methods for higher-index DAEs start with *index reduction*: augmenting the DAE by time-derivatives of some of its equations to produce a DAE of larger size and smaller index. Various index reduction methods have been used that convert the DAE to an ODE with more degrees of freedom than the DAE. Then the solution paths of the DAE are a proper subset of those of the ODE. This tends to be bad numerically, as errors cause drift from the consistent manifold that is often exponential once it starts.

Dummy derivatives (DDs) by contrast are a systematic way to form an equivalent ODE with *exactly as many* DOF as the DAE. If one regards the DAE as a flow on the consistent manifold \mathcal{M} , DDs describes the flow in a local coordinate system for \mathcal{M} . Thus numerical drift can only be within \mathcal{M} , where it is less harmful. However if the path leaves the patch of \mathcal{M} where the coordinate system is nonsingular, one must choose new coordinates. This need for *DD switching*, or *pivoting*, complicates a numerical algorithm.

The following description of the DDs process is equivalent to that in [7]. The set of possible matrix sequences (\mathbf{G}_k) whenever one selects a state vector, below, is the same in either method, but we find \mathbf{G}_k from smallest up (each is a sub-matrix of the next), while [7] finds them in the opposite order.

Assume c_i and d_j are the canonical offsets. First form the derivatives of each $f_i = 0$ up to the c_i th, forming the augmented system of $N_f = n + \sum_i c_i$ equations:

$$f_i^{(l)} = 0, \quad l = 0, \dots, c_i, \quad i = 1, \dots, n. \quad (8)$$

Its unknowns are the $N_x = n + \sum_j d_j$ derivatives of the state variables x_j up to the d_j th. View them for now as unrelated algebraic unknowns that we call *items*, and to emphasise this denote them x_{jl} :

$$x_{jl} \text{ renames } x_j^{(l)}, \quad l = 0, \dots, d_j, \quad j = 1, \dots, n. \quad (9)$$

The system has fewer equations than variables by the amount $\sum_j d_j - \sum_i c_i$, which equals the number DOF of degrees of freedom. To balance this, DDs finds a number DOF of items x_{jl} to be *state items*, for (j, l) in a suitable set S of index pairs, chosen such that all the

other items can locally be solved for as functions of these. The *state vector* \mathbf{x}_S formed by the state items is the associated local coordinate system of the manifold \mathcal{M} .

One requires $l < d_j$ for each $(j, l) \in S$, so that $x_{j,l+1}$ is also an item. Then the differential relations between each state item and its next higher derivative:

$$\dot{x}_{jl} = x_{j,l+1} \quad (10)$$

can be interpreted as an ODE system for the state items.

State vector selection—initially or at a DD-switching point—may be done as follows. The $n \times n$ system Jacobian \mathbf{J} in (6) is nonsingular there. For $k = k_d, k_d + 1, \dots, -1$ where k_d is minus the largest d_j , the “standard solution scheme” of the Σ -method constructs sub-matrices \mathbf{J}_k of \mathbf{J} by selecting those rows i where $k + c_i \geq 0$ and columns j where $k + d_j \geq 0$. Then: \mathbf{J}_k is of full row rank; it has size $m_k \times n_k$ where $m_k \leq n_k$; the sum of the differences $\sum_k (n_k - m_k)$ equals DOF. For each k , select m_k columns of \mathbf{J}_k that form a nonsingular matrix \mathbf{G}_k . This can and must be done in such a way that the set of selected columns increases with k , so that each \mathbf{G}_k is a sub-matrix of the next. For each of the $(n_k - m_k)$ *unselected* columns j consider the item $x_j^{(k+d_j)}$. The set of all these is a valid state vector \mathbf{x}_S since, briefly, non-singularity of \mathbf{G}_k ensures that at stage k , “selected” items $x_j^{(k+d_j)}$ belonging to selected columns can, by the Implicit Function Theorem, be found locally as functions of the unselected items.

As said, (10) thus becomes a size-DOF ODE system,

$$\dot{\mathbf{x}}_S = \mathbf{F}(t, \mathbf{x}_S). \quad (11)$$

This is locally equivalent to the size- N_x DAE (8, 10) and hence to the original DAE. Though “index-1” is the usual term used, the stronger property holds that

$$(8, 10) \text{ form an } \textit{implicit ODE},$$

defined as an SA-friendly DAE whose offsets c_i are all zero.

2.2 Practical considerations

At a DD-switch, the set (8) of differentiated equations does not change. Thus at the housekeeping level, a switch merely changes the set S of index pairs (j, l) that define the state vector.

We verified that the method works, by a proof-of-concept MATLAB implementation. One example was the double pendulum (one pendulum-rod hung off another) in x, y coordinates, where each rod independently has four DD-switching points in a full rotation, one in each of the four quadrants, giving $4 \times 4 = 16$ possible “DD modes”.

It remains to be seen however how efficient one can make DD-switching for production code and for larger problems. Finding the \mathbf{G}_k at a switch is nontrivial. Ideally one wants each one to be maximally well-conditioned, which is expensive, so one seeks heuristic methods. For this reason Scholz and Steinbrecher’s simplified method [15] is interesting. It finds a state vector based on a highest-value transversal of the signature matrix; it is less general than full DDs but cheaper. One might try it first, and if it gives ill-conditioned \mathbf{G}_k then use full DDs.

It seems natural to solve the original DAE numerically by giving formulation (8, 10) to a standard index-1 DAE solver. However many models, especially mechanical ones, have many equations but few degrees of freedom, $N_x \gg \text{DOF}$. Then it makes sense to

convert to the explicit form (11). In many mechanical contexts (though not all) this ODE is typically non-stiff and thus amenable to solving by, say, an explicit Runge–Kutta method. Working memory for sub-problems of size up to n is needed by the root-finding that forms (11), but is typically less than that needed by an implicit DAE code on a problem of size N_x . For more discussion see [13].

2.3 Example

EXAMPLE 2.1 (Pendulum) Let the original DAE be the simple pendulum in cartesian coordinates, shown with its signature matrix (7), with relevant transversals marked. Gravity g and length ℓ are constants, and $x(t)$, $y(t)$ and $\lambda(t)$ are state variables.

$$\begin{aligned} 0 &= A = \ddot{x} + x\lambda, \\ 0 &= B = \ddot{y} + y\lambda - g, \\ 0 &= C = x^2 + y^2 - \ell^2, \end{aligned} \quad \Sigma = \begin{array}{ccccc} & x & y & \lambda & c_i \\ A & 2^\bullet & & 0^\circ & 0 \\ B & & 2^\circ & 0^\bullet & 0 \\ C & 0^\circ & 0^\bullet & & 2 \\ d_j & 2 & 2 & 0 & \end{array} \quad (12)$$

To describe DDs here, we modify the general x_{jl} notation by renaming x, \dot{x}, \ddot{x} to x_0, x_1, x_2 , and so on, to get 5 equations in 7 unknowns:

Augmented system	After renaming	
$0 = A = \ddot{x} + x\lambda$	$0 = A_0 = x_2 + x_0\lambda_0$	
$0 = B = \ddot{y} + y\lambda - g$	$0 = B_0 = y_2 + y_0\lambda_0 - g$	
$0 = C = x^2 + y^2 - \ell^2$	$0 = C_0 = x_0^2 + y_0^2 - \ell^2$	
$0 = \dot{C} = 2(x\dot{x} + y\dot{y})$	$0 = C_1 = 2(x_0x_1 + y_0y_1)$	(13)
$0 = \ddot{C} = 2(x\ddot{x} + \dot{x}^2 + y\ddot{y} + \dot{y}^2)$	$0 = C_2 = 2(x_0x_2 + x_1^2 + y_0y_2 + y_1^2)$	
unknowns $x, \dot{x}, \ddot{x}, y, \dot{y}, \ddot{y}, \lambda$	unknowns $x_0, x_1, x_2, y_0, y_1, y_2, \lambda_0$	

One can choose any of $(x, \dot{x}), (y, \dot{y}), (x, \ddot{x}), (y, \ddot{y})$ as state vector (one *must* choose one undifferentiated variable and one first derivative), but only the first two are “convenient” for AD, as the next section shows.

Suppose for example $\mathbf{x}_S = (x, \dot{x}) = (x_0, x_1)$. It is easily seen that provided $y \neq 0$ one can find all the items as functions of these two, hence the pendulum DAE is equivalent to an ODE (11) in this \mathbf{x}_S when $y \neq 0$.

The description of DDs given in §2.1 has the advantage of combining *index reduction* and *order reduction* into one process. For computer solution, it is probably easiest to work with the order 1 DAE formed by the $N_x = n + \sum_j d_j$ equations (8, 10). However “by hand”, one can simplify by directly substituting the derivative relations into (8) where possible. E.g., when \mathbf{x}_S is (x_0, x_1) , one obtains

$$\begin{aligned} 0 &= A_0 = x_1 + x_0\lambda_0 \\ 0 &= B_0 = y_2 + y_0\lambda_0 - g \\ 0 &= C_0 = x_0^2 + y_0^2 - \ell^2 \\ 0 &= C_1 = 2(x_0x_1 + y_0y_1) \\ 0 &= C_2 = 2(x_0\dot{x}_1 + x_1^2 + y_0y_2 + y_1^2) \\ 0 &= \dot{x}_0 - x_1 \end{aligned}$$

The last equation, $\dot{x}_0 = x_1$, can not be “substituted away”—in general, any equation (10) must stay if its x_{jl} and $x_{j,l+1}$ are both state items, as this is how order reduction occurs.

In [7]’s terminology a “dummy derivative” means a differentiated item that, in our terms, is solved for item but is not a state variable or the derivative of one. In this example with this state vector, that makes y_1 and y_2 the DDs. ■

2.4 AD for dummy derivatives

How can an AD tool help automate numerical solution by DDs, as described above?

First, it is helpful if the tool supports d/dt as a first-class operator, of equal status with $+$, \times , $\sin()$, etc., so that it can understand a representation of a DAE in the general form (1). This is not essential. Tools such as ADOL-C and dcc/dco [4, 8] do not have this feature, but can handle arbitrary expressions containing derivatives by renaming the latter as algebraic items and stating their differential relations separately. This is like the method in §2.1, where derivatives are renamed as algebraic in (9) and some differential relations between them stated in (10).

Our solver DAETS uses Ole Stauning’s AD package FADBAD++ [17], written in C++. It did not originally include d/dt but at our request in 2002, Stauning included the operator `Diff` such that `Diff(\cdot , q)` means d^q/dt^q . For instance, straightforward code for the pendulum, as in the DAETS user guide, is shown in Figure 1.

```

1  template <typename T>
2  void fcn(T t, const T *z, T *f, void *param) {
3      // z[0], z[1], z[2] are x, y, λ.
4      const double G = 9.81, L = 10.0;
5      f[0] = Diff(z[0], 2) + z[0]*z[2];
6      f[1] = Diff(z[1], 2) + z[1]*z[2] - G;
7      f[2] = sqr(z[0]) + sqr(z[1]) - sqr(L);
8  }
```

Figure 1. Code for simple pendulum problem.

More important, for DDs and other index reduction methods, an AD tool must be able to differentiate the f_i selectively. For instance in the pendulum, A and B are to be left alone, and C differentiated twice.

At first sight this seems to require a tool based on source code transformation, which could generate code symbolically for the last two equations in (13), for instance. But this is not the case—the key is not to treat different derivatives of a given variable in isolation, but store them together as a truncated power series. For instance in the pendulum, the unknowns form three objects

$$\begin{array}{ll}
x &= (x_0, x_1, x_2) \quad \text{order 2 power series,} \\
y &= (y_0, y_1, y_2) \quad \text{order 2 power series,} \\
\lambda &= (\lambda_0) \quad \text{order 0 power series.}
\end{array}$$

To keep the algebra simple below, these are Taylor coefficients, not derivatives, thus x_k here is the x_k in (13) divided by $k!$, and so on.

Taylor series AD by overloading, provided by many AD tools, now gives the needed

values. For instance evaluating $C = x^2 + y^2 - \ell^2$ proceeds via these intermediate steps:

input			
x		$= (x_0, x_1, x_2)$	
y		$= (y_0, y_1, y_2)$	
<hr/>			
compute			
v_1	$= x^2$	$= (x_0^2, \quad 2x_0x_1, \quad 2x_0x_2 + x_1^2)$	
v_2	$= y^2$	$= (y_0^2, \quad 2y_0y_1, \quad 2y_0y_2 + y_1^2)$	
v_3	$= v_1 + v_2$	$= (x_0^2 + y_0^2, \quad 2(x_0x_1 + y_0y_1), \quad 2(x_0x_2 + y_0y_2) + x_1^2 + y_1^2)$	
C	$= v_3 - \text{const}(\ell^2)$	$= (x_0^2 + y_0^2 - \ell^2, \quad 2(x_0x_1 + y_0y_1), \quad 2(x_0x_2 + y_0y_2) + x_1^2 + y_1^2)$	

returning an order 2 power series object C holding the needed coefficients (C_0, C_1, C_2) , that is $(C, \dot{C}, \frac{1}{2}\ddot{C})$ in terms of derivatives.

Evaluating $A = \ddot{x} + x\lambda$ and $B = \ddot{y} + y\lambda - g$ is similar. `Diff(·, 2)` converts, e.g., the order 2 series $x = (x_0, x_1, x_2)$ to the order 0 series $(2x_2)$. Thus A, B are returned as order 0 series $A = (A_0) = (2x_2 + x_0\lambda_0)$, $B = (B_0) = (2y_2 + y_0\lambda_0 - g)$.

The above method gives an explicit evaluation of the N_f functions (8) at the N_x arguments (9).

In the context of DDs and reducing the DAE to an explicit ODE, state item values, say values $\mathbf{x}_S = (x_0, x_1)$ are given; the 5 items $\mathbf{x}_F = (x_2, y_0, y_1, y_2, \lambda_0)$ are trial values that produce 5 residual values $\mathbf{r} = (A_0, B_0, C_0, C_1, C_2)$. By root-finding using suitable Jacobians (found by methods not described here) we find \mathbf{x}_F that makes $\mathbf{r} = \mathbf{0}$, thus solving for \mathbf{x}_F as a function of \mathbf{x}_S . Extract x_2 from \mathbf{x}_F to form (x_1, x_2) , which is $\dot{\mathbf{x}}_S$. This implements the function \mathbf{F} in (11).

To make this work, the state items must comprise a contiguous set of derivatives of each variable, with no gaps. (Hence, see below (13), (x, \dot{y}) and (y, \dot{x}) are not useful state vectors for the pendulum.) That is, S must have the form $\{(j, l) \mid 0 \leq l < \delta_j, j = 1, \dots, n\}$, where $\boldsymbol{\delta} = (\delta_1, \dots, \delta_n)$ is an integer *DD-spec vector* with $0 \leq \delta_j \leq d_j$ and $\sum_j \delta_j = \text{DOF}$, which uniquely specifies the DD scheme currently in use. DD switching can be based on changing this $\boldsymbol{\delta}$, and following through the consequences for various associated index sets and Jacobian-related matrices.

Experiments by Nedialkov confirm this is an effective and flexible way to implement DDs, including the root-finding that produces (11), using methods already provided by the DAETS classes.

3. The Lagrangian

3.1 Lagrangian mechanics theory

For mechanical systems, such as in robotics, equations of motion can often be conveniently derived from the system's Lagrangian function L . It is assumed there are conservative (energy preserving) forces such that one can define a potential energy V depending only on system position. Then $L = T - V$, where T is the system's total kinetic energy. Let the configuration at any time be described by generalised coordinates $\mathbf{q} = (q_1, \dots, q_{n_q})$ such that T is a function of $\dot{\mathbf{q}}$ and possibly \mathbf{q} , and V is a function of \mathbf{q} only. There may (depending on the coordinate system used) be n_c scalar constraints that are holonomic, i.e. functions of positions and possibly time but not of velocities, namely $C_j(t, \mathbf{q}) = 0$.


```

1  template <typename T>
2  void fcn( T t, const T *z, T *f, void *param ) {
3      vector< B<T> > q(SIZEOFQ), qp(SIZEOFQ), C(SIZEOFC);
4      init_q_qp(z, q, qp);
5      double *p = (double *)param;
6      double m = p[0], g = p[1], l = p[2];
7      B<T> x = q[0], y = q[1], xp = qp[0], yp = qp[1];
8      B<T> L = 0.5 * m * (sqr(xp) + sqr(yp)) + m * g * y;
9      C[0] = sqr(x) + sqr(y) - sqr(l);
10     setupEquations(L, z, q, qp, C, f);
11 }

```

Figure 2. Code to describe pendulum in Lagrangian form.

Then the variational principle of stationary action gives the (n_q+n_c) equations of motion:

$$\frac{d}{dt} \frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} + \sum_{j=1}^{n_c} \lambda_j \frac{\partial C_j}{\partial q_i} = 0, \quad i = 1, \dots, n_q, \quad (14)$$

$$C_j(t, \mathbf{q}) = 0, \quad j = 1, \dots, n_c, \quad (15)$$

where the λ_j are Lagrange multipliers for the constraints. If $n_c > 0$, i.e. constraints are present, (14, 15) is termed a Lagrangian system *of the first kind*. It is a DAE system, of index 3 in the classical sense or index 2 as defined in (5), since two t -differentiations of each C_j are needed. If the coordinates are chosen so that $n_c = 0$, it is *of the second kind* and is an ODE system.

For a system subject to external forces, the zero right hand sides of (14) are replaced by $u_i(t)$, $i = 1, \dots, n_q$, which are *generalised external force components*. One can model certain kinds of internal dissipative (not energy conserving) velocity-dependent force by a so called *Rayleigh dissipative term*.

For instance, for free motion of the simple pendulum, taking \mathbf{q} to be (x, y) , the coordinates of the pendulum bob with y measured downward, gives

$$L = \frac{1}{2}m(\dot{x}^2 + \dot{y}^2) + mgy, \quad (16)$$

where m is the bob's mass, and with one constraint $x^2 + y^2 - \ell^2 = 0$. Then (14, 15) lead to equations (12). But taking \mathbf{q} to be (θ) , the angle of the pendulum from the downward vertical, gives $L = \frac{1}{2}m(l\dot{\theta})^2 + mgl \cos \theta$, with no constraints. Then (14, 15) lead to the ODE $\ddot{\theta} = -(g/l) \sin \theta$, which is equivalent to (12).

Our method applies to solution by the C++ code DAETS. We describe how it looks to the user, not the inner workings. The user describes a DAE system to DAETS using a template type T , as Figure 1 in §2.4 shows. DAETS instantiates T with several concrete types during execution to perform structural analysis and other tasks—in particular Taylor coefficient generation during numerical solution using FADBAD++'s Taylor type.

The Lagrangian facility overlays the FADBAD++ backward (reverse mode) differentiation type B on top of T to create the type $B<T>$ which is then suitably manipulated. It avoids the seemingly needed two “phases”: the equations of motion are created, and readied for numerical solution, in one operation that happens at runtime.

The pendulum in x, y form may be coded as the `fcn` function in Figure 2 that defines a DAE in the form accepted by DAETS. Here `SIZEOFQ` and `SIZEOFC` are macros, set

to 2 and 1. The physical parameters m , g and ℓ are passed in the array `param`. DAETS specifies this as having type `void`, which leaves the user free to use some other type than `double` if needed. (Since m is a factor all through L , it is superfluous and could have been omitted.)

Lines 5–7 name things for readability; lines 8–9 then define the problem. Lines 3–4 and 10 are boilerplate and do not change from one problem to the next. The call `init_q_qp` “connects” the input variables of (12) in z to the `q` and `qp` variables and prepares them for the partial derivatives computation. The `setupEquation` call sets up the AD for converting the form (16) to (12). When `setupEquation` returns, `f[0]`, `f[1]`, and `f[2]` contain A , B and C in (12), (modulo a factor m).

3.2 Examples

We have applied the DAETS Lagrangian facility to various systems, including the following examples. Performance tests are on a 2015 MacBook Pro laptop with a 4-core 2.2 GHz Intel processor running Mac OS X 10.11.6. The C++ compiler is GCC 6.3.0. All numerics are in C++ `double`.

Nedialkov’s group at McMaster University have recently improved substantially the performance of the AD in the Taylor series method of FADBAD++ by applying common subexpression elimination techniques. They have also implemented an efficient algorithm for computing the System Jacobians, propagating compressed gradients, and have incorporated sparse linear algebra in DAETS. Speedups of > 10 compared to using the original FADBAD++ are common. As a result code created by the Lagrangian facility is surprisingly fast.

Because of the perceived difficulty of solving DAEs, generalised coordinates are often chosen to eliminate the constraints and give a Lagrangian of the second kind. For instance a rigid body’s 3D position can be described by 3 coordinates of the position of its centre of mass and 3 of its angular position relative to this. However the mathematical formulation is often simpler in cartesian coordinates. One plus of using a code for high-index DAEs such as DAETS is that it handles resulting “first kind” systems easily. Further, since DAETS does not set up a local coordinate system for numerical solution as the DDs method does, it does not suffer the performance penalty of DD-switching.

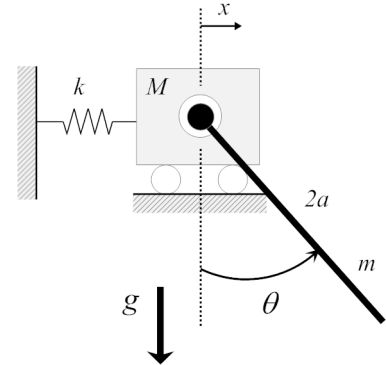


Figure 3. Spring-Mass-Pendulum with one rod.

EXAMPLE 3.1 (Spring-Mass-Pendulum) This 2D model is taken from an article on the Acumen mechanics modeling system by Zhu, Taha *et al.* [19].

We have extended their model to a chain of any number n of rods. Namely, a horizontally sliding point-mass M is connected by a spring of stiffness k to a fixed point at the same level. From M hangs a chain of n uniform rods, with frictionless joints between the end of one and the start of the next. Purely to simplify the code, they all have the same mass m and length $l = 2a$. We assume the setup is constructed so that all components can slide or rotate freely without colliding.

For $n \geq 2$ (possibly even for $n = 1$) the motion can be chaotic. The figure (taken from [19]) shows the case $n = 1$. As the figure indicates, [19] takes $\mathbf{q} = (x, \theta)$ as coordinates,

leading to a Lagrangian of the second kind, $L = T - V$ where:

$$T = \frac{1}{2}(M + m)\dot{x}^2 + m\dot{x}\dot{\theta}\cos\theta + \frac{2}{3}ma^2\dot{\theta}^2, \quad V = \frac{1}{2}kx^2 + mga(1 - \cos\theta). \quad (17)$$

Here the rotational kinetic energy term $\frac{2}{3}ma^2\dot{\theta}^2$ uses the moment of inertia $I = \frac{4}{3}ma^2$ of a uniform rod about its centre of mass.

For the general n -rod model we use, instead, cartesian coordinates $\mathbf{q} = (x_0, x_1, \dots, x_n; y_1, \dots, y_n)$. Here $\mathbf{r}_0 = (x_0, y_0)$, with y_0 constant equal to 0, is the position of M and the start of rod 1, and $\mathbf{r}_i = (x_i, y_i)$ for $i = 1, \dots, n$ is the position of the joint between the end of rod i and (for $i < n$) the start of rod $i + 1$. We avoid moments of inertia by using the following, where \cdot denotes the dot product of vectors.

LEMMA 3.1 *If the ends of a uniform rod of mass m have position vectors \mathbf{r}_0 and \mathbf{r}_1 , depending on t , then its kinetic energy at any instant is*

$$\text{KE} = \frac{1}{6}m(\dot{\mathbf{r}}_0 \cdot \dot{\mathbf{r}}_0 + \dot{\mathbf{r}}_0 \cdot \dot{\mathbf{r}}_1 + \dot{\mathbf{r}}_1 \cdot \dot{\mathbf{r}}_1).$$

Proof. We can parameterise position along the rod as $\mathbf{r} = (1 - s)\mathbf{r}_0 + s\mathbf{r}_1$, for $0 \leq s \leq 1$. Since the rod has total mass m , an element from s to $s + ds$ has mass $m ds$. The velocity of this element is $\dot{\mathbf{r}} = (1 - s)\dot{\mathbf{r}}_0 + s\dot{\mathbf{r}}_1$ so its kinetic energy is

$$\frac{1}{2}m(\dot{\mathbf{r}} \cdot \dot{\mathbf{r}})ds = \frac{1}{2}m((1 - s)^2 \dot{\mathbf{r}}_0 \cdot \dot{\mathbf{r}}_0 + 2(1 - s)s \dot{\mathbf{r}}_0 \cdot \dot{\mathbf{r}}_1 + s^2 \dot{\mathbf{r}}_1 \cdot \dot{\mathbf{r}}_1) ds.$$

Integrating this from 0 to 1 gives the result. ■

The potential energy of the rods comes from considering the mass of rod i to be at its centre of mass at height $\frac{1}{2}(y_{i-1} + y_i)$; there is a contribution of $\frac{1}{2}kx_0^2$ from the spring and none from mass M . This leads to the Lagrangian $L = T - V$, and constraints C_i , where

$$\begin{aligned} T &= \frac{1}{2}M\dot{x}_0^2 + \frac{1}{6}m \sum_{i=1}^n (\dot{\mathbf{r}}_{i-1} \cdot \dot{\mathbf{r}}_{i-1} + \dot{\mathbf{r}}_{i-1} \cdot \dot{\mathbf{r}}_i + \dot{\mathbf{r}}_i \cdot \dot{\mathbf{r}}_i) \\ &= \frac{1}{2}M\dot{x}_0^2 + \frac{1}{6}m \sum_{i=1}^n ((\dot{x}_{i-1}^2 + \dot{y}_{i-1}^2) + (\dot{x}_{i-1}\dot{x}_i + \dot{y}_{i-1}\dot{y}_i) + (\dot{x}_i^2 + \dot{y}_i^2)) \\ V &= \frac{1}{2}kx_0^2 + mg \sum_{i=1}^n \frac{1}{2}(y_{i-1} + y_i) = \frac{1}{2}kx_0^2 + mg\left(\frac{1}{2}y_n + \sum_{i=1}^{n-1} y_i\right), \\ 0 &= C_i = (x_i - x_{i-1})^2 + (y_i - y_{i-1})^2 - \ell^2, \quad (i = 1, \dots, n). \end{aligned}$$

The code in Figure 4, which replaces lines 7–9 in the `fcn` of Figure 2, implements the above formulas. Here n , the number of rods, is read in as one of the physical parameters. `SIZEOFC` also equals n . The arrays `q` and `qp` holding \mathbf{q} and $\dot{\mathbf{q}}$ have length $2n + 1$.

Listing line 1 uses C syntax¹ to split `q` into a scalar holding x_0 , and two size- n arrays holding x_1, \dots, x_n , and y_1, \dots, y_n ; similarly `qp`. The variable `KEsum` accumulates $\dot{x}_0^2 + \dot{x}_n^2 + \dot{y}_n^2 + \dot{x}_0\dot{x}_1 + \sum_{i=1}^{n-1} [2(\dot{x}_i^2 + \dot{y}_i^2) + \dot{x}_i\dot{x}_{i+1} + \dot{y}_i\dot{y}_{i+1}]$, which is equivalent to the sum in T , and similarly `PEsum`.

In the computation of L , the temporary dependent variables `KEsum`, `PEsum`, `KE`, and `PE`, are local in the block between lines 3 and 14; `FADBAD++` requires that in the reverse

¹For instance `q+1` references a sub-array of `q` starting at `q[1]`.

```

1  B<T> x0 = q[0], *x = q+1, *y = x+n, x0p = qp[0], *xp = qp+1, *yp = xp+n;
2  B<T> L;
3  {
4      B<T> KESum = sqr(x0p) + sqr(xp[n-1]) + sqr(yp[n-1]) + x0p*xp[0];
5      for (int i=0; i < n-1; i++)
6          KESum += 2*( sqr(xp[i]) + sqr(yp[i]) ) + xp[i]*xp[i+1] + yp[i]*yp[i+1];
7      B<T> KE = 0.5*M*sqr(x0p) + m/6*KESum;
8
9      B<T> PESum = 0.5*y[n-1];
10     for (int i=0; i<n-1; i++) PESum += y[i];
11     B<T> PE = 0.5*k*sqr(x0) - m*g*PESum;    // - as y goes downward
12
13     L = KE - PE;
14 }
15 C[0] = sqr(x[0]-x0) + sqr(y[0]) - sqr(l);
16 for (int i=1; i<SIZEOFC; i++)
17     C[i] = sqr(x[i]-x[i-1]) + sqr(y[i]-y[i-1]) - sqr(l);

```

Figure 4. Code for spring-mass-multi-pendulum system.

tol	$n=1$	2	4	6	8	10	12	14	16	18	20
1e-4	0.19	0.44	1.06	2.98	6.12	8.94	12.10	19.75	29.72	39.83	47.96
1e-6	0.23	0.56	1.45	4.49	8.11	11.03	17.70	25.28	37.37	53.56	65.68
1e-8	0.34	0.68	1.83	5.76	10.84	14.94	24.13	35.42	51.12	73.42	86.13
1e-10	0.40	0.91	2.63	7.41	14.92	19.02	33.29	44.75	67.31	98.79	121.5
1e-12	0.55	1.35	3.59	9.99	18.40	26.89	45.90	62.79	96.56	124.5	155.9

Table 1. Time (seconds) to integrate to $t = 100$ for various numbers n of rods, and tolerances `tol`.

mode either all intermediate dependent variables are differentiated or go out of scope, which is the case here.

In our tests the physical parameters of the original model in [19] were used, namely, assuming SI units, $g = 9.8 \text{ m s}^{-2}$, $l = 2a = 2 \text{ m}$, $M = 5 \text{ Kg}$, $m = 2 \text{ Kg}$, $k = 10 \text{ Kg s}^{-2}$.

The chosen initial conditions (ICs) are that the system is at rest with mass M at $x = 4$, and the rods stretched horizontally to the left. (Thus the spring is pushing against the row of rods; animations show it “folds up” rods 1 and 2 as they start to fall.)

To confirm that we are modeling the same system as in [19], the equations of motion derived from the Lagrangian (17) given in [19] were coded in MATLAB and integrated by `ode45`. The results were compared with those of the DAETS version for the case $n = 1$. The latter was coded to output q and \dot{q} at each of its time points t_i . This data was mapped to the t_i chosen by the MATLAB version by Hermite cubic interpolation between adjacent t_i of DAETS. Figure 5 shows that over $t = [0, 40]$, the differences (`ode45` solution at tolerance 10^{-12}) – (DAETS solution at tolerance 10^{-8}) are of order 10^{-6} . This gives confidence that the programs are solving the same physical model.

For timing tests, the system was integrated by DAETS over $0 \leq t \leq 100$ for various numbers n of rods and (mixed relative-absolute) tolerances `tol`. Case $n = 1$ is the model in [19]. The Taylor series order was set to 15, which works well for these problems at this range of accuracies. Table 1 shows the times taken.

DAETS has a “maximum step size” feature but this was not used so it chose the step sizes h freely. For the “hardest” problem $n = 20$ at tolerance 10^{-12} they ranged from $h = 0.00061$ to $h = 0.093$. For the “easiest”, $n = 1$ at tolerance 10^{-4} , they ranged from $h = 0.065$ to $h = 0.421$. ■

EXAMPLE 3.2 (Controlled simple pendulum) We show one can solve a prescribed-

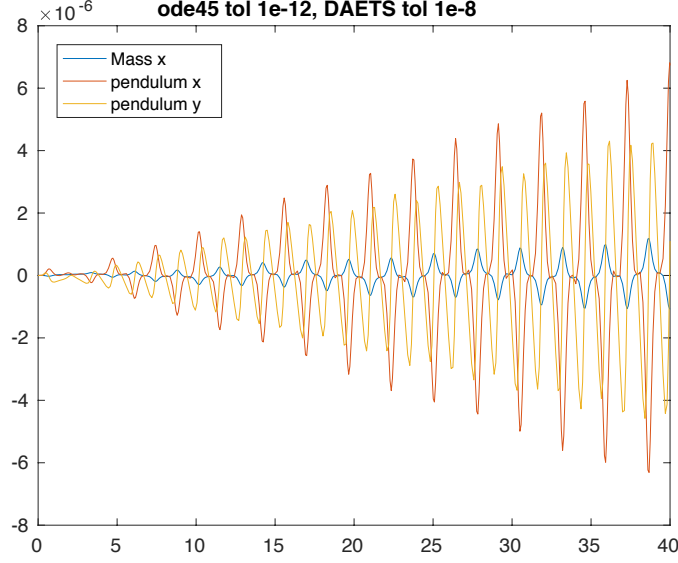


Figure 5. Spring-mass-pendulum with $n = 1$. It shows, for the x coordinate of the sliding mass and the x and y coordinates of the pendulum, the difference between the solution by our model and that by (17), over $0 \leq t \leq 40$.

trajectory control problem for a Lagrangian-described system. Namely, for the simple pendulum we introduce a horizontal external force on the bob, modeled as a system input $u = u(t)$ such that the equation $\ddot{x} + \lambda x = 0$ becomes

$$\ddot{x} + \lambda x - u = 0. \quad (18)$$

The aim is to find $u(t)$ (plus suitable consistent ICs) so that the x position performs simple harmonic motion $x(t) = a \sin(\omega t)$ exactly, where the constants a and ω are a given amplitude and frequency, respectively.

Comparing the pendulum as initial-value problem in Figure 2 and as control problem in Figure 6 shows the *implementation* changes little. One passes a and ω as extra parameters that become a and w . After the `setupEquations` line, the first equation `f[0]` is modified in line 12, and a new fourth equation `f[3]` is at line 13 (in which the x at line 7 in Figure 2 cannot be used as it has the wrong type, `B<T>` instead of `T`).

But the revision has changed the DAE's mathematical nature greatly. Now with 4 variables and equations, it is shown below with its signature matrix Σ (a blank means $-\infty$, and the unique transversal is marked by $^\circ$).

$$\begin{aligned} 0 = A &= x'' + x\lambda - u \\ 0 = B &= y'' + y\lambda - g \\ 0 = C &= x^2 + y^2 - \ell^2 \\ 0 = D &= x - a \sin(\omega t) \end{aligned} \quad \Sigma = \begin{array}{ccccc} & x & y & \lambda & u & c_i \\ A & 2 & & 0 & 0^\circ & 0 \\ B & & 2 & 0^\circ & & 0 \\ C & 0 & 0^\circ & & & 2 \\ D & 0^\circ & & & & 2 \\ d_j & 2 & 2 & 0 & 0 & \end{array} \quad (19)$$

While (12) has 2 degrees of freedom, (19) has none—specifying the desired $x(t)$ determines the system input $u(t)$, as well as y and λ , uniquely.

With the physical parameters $g = 9.8$ and $\ell = 10$, the problem was solved by DAETS with ω equal to the pendulum's natural frequency $\sqrt{g/\ell}$ of small oscillations, and for

```

1  template <typename T>
2  void fcn( T t, const T *z, T *f, void *param ) {
3      vector< B<T> > q(SIZEOFQ), qp(SIZEOFQ), C(SIZEOFC);
4      init_q_qp(z, q, qp);
5      double *p = (double *)param;
6      double m = p[0], g = p[1], l = p[2],
7             a = p[3], w = p[4];
8      B<T> x = q[0], y = q[1], xp = qp[0], yp = qp[1];
9      B<T> L = 0.5 * m * (sqr(xp) + sqr(yp)) + m * g * y;
10     C[0] = sqr(x) + sqr(y) - sqr(l);
11     setupEquations(L, z, q, qp, C, f);
12     f[0] = z[3];
13     f[3] = z[0] - a*sin(w*t);
14 }

```

Figure 6. The fcn for the controlled pendulum problem: new lines 12 and 13 are inserted.

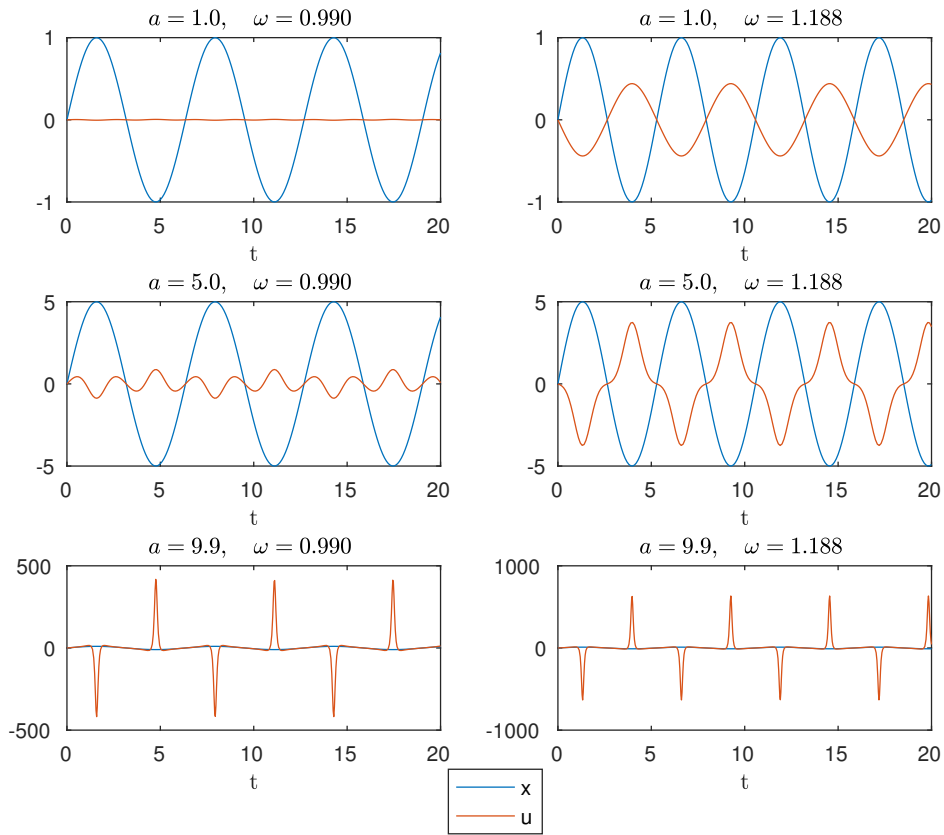


Figure 7. Solution by DAETS of system input $u(t)$ for controlled pendulum with $g = 9.8$, $l = 10$. Required response $x = a \sin(\omega t)$. For ω equal to natural angular frequency (left column) and 20% larger (right column), and three a values.

various a ; and again with ω changed by 20%, for the same a values. Some examples of resulting u 's are plotted over several cycles in Figure 7. As expected, u is very small when ω is the natural frequency and a is small. It becomes large as a approaches ℓ , or as the frequency moves away from the natural one. DAETS took less than 0.1 seconds for each of the runs. ■

EXAMPLE 3.3 (DETEST Non-stiff Problem C5) This problem from the non-stiff part of the DETEST testing package for ODE solvers [2], and originally² from Zonneveld [20], is titled “Five Body Problem: Motion of five outer planets about the Sun”. It is a order 2 ODE of size 15 (so size 30 when reduced to order 1), the variables being the positions of Jupiter, Saturn, Uranus, Neptune and Pluto relative to the Sun, in x, y, z coordinates such that the ecliptic plane, in which the orbits approximately lie, is not close to any of the three axes.

To set up the Lagrangian formulation, \mathbf{q} comprising the 5 relative positions $(\mathbf{q}_1(t), \dots, \mathbf{q}_5(t))$ (each \mathbf{q} being a 3-vector) is converted to 6 positions $(\mathbf{r}_0(t), \dots, \mathbf{r}_5(t))$ of Sun and planets relative to their common centre of mass, which may be considered to be at rest in a Newtonian absolute frame. Namely let m_0 be the mass of the Sun and m_1, \dots, m_5 the masses of the planets and subtract

$$\mathbf{r}_c = \frac{m_0 \mathbf{0} + (m_1 \mathbf{q}_1 + \dots + m_5 \mathbf{q}_5)}{m_0 + (m_1 + \dots + m_5)}$$

from each component of $(\mathbf{0}, \mathbf{q}_1, \dots, \mathbf{q}_5)$ to get $(\mathbf{r}_0, \dots, \mathbf{r}_5)$. Then

$$T = \frac{1}{2} \sum_{i=0}^5 m_i |\dot{\mathbf{r}}_i|^2, \quad V = - \sum_{i=0}^5 \sum_{j=i+1}^5 \frac{G m_i m_j}{|\mathbf{r}_i - \mathbf{r}_j|}, \quad L = T - V, \quad (20)$$

where G is the gravitational constant. The code, shown in Appendix A, was made particularly compact using a C++ 3-vector class from [3].

In the DETEST model the time unit (TU) is 100 days. Distance is measured in astronomical units (AU), where 1 AU is the mean radius of the earth’s orbit. The task is to integrate from given initial values up to $t = 20$ TU; at tolerance 10^{-13} we get agreement with DETEST’s reference solution to around 12 decimal places.

To see how fast the solution is, the problem was integrated to $t = 200,000$ TU (about 55,000 earth years), with Taylor order 15, at tolerances 10^{-13} and 10^{-14} . The two sets of results agree to five decimal places, and some DAETS integration statistics are

Integration of Sun and 5 planets to $t = 200,000$ TU				
tol	CPU secs	no. of steps	smallest step	largest step
1e-13	31.2	58028	1.996	5.654
1e-14	36.2	67035	1.725	5.370

It is known that Pluto is locked in a, currently stable, 3:2 resonance with Neptune. This was easy to verify over short periods from our results. For the subtleties of solar system behaviour see [5] and references therein. This article cites evidence that over very long times the system switches between regular and chaotic behaviour in an irregular way that depends critically on ICs. Hence numerical results showing linear (regular) divergence of neighbouring solutions up to some large time T —rather than exponential (chaotic) divergence—are no evidence that such behaviour will continue up to, say, time $2T$.

What about the given ICs? We integrated the problem at two tolerances 1e-13 and 1e-15, recording the solutions at successive powers of ten up to 10^8 TU (< 5 hours CPU time for each to reach 10^8) and computing the relative error in the two norm at these times. The results, see Figure 8, indicate non-chaotic behaviour up to that point. ■

²[2] cites nonexistent reference “11” which should be “10” and is the Zonneveld work.

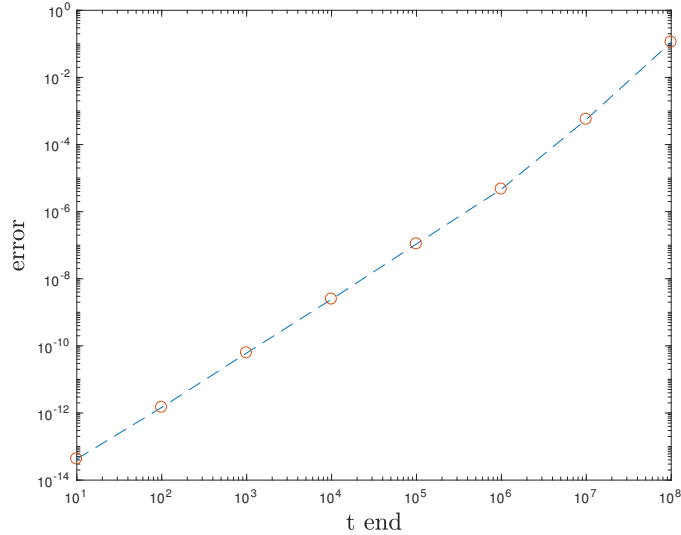


Figure 8. Log-log plot of divergence vs. time. The planet model was integrated at two tolerances $1e-13$ and $1e-15$. The norm of the difference between these solutions is plotted at 10^n TU, $n = 1, \dots, 8$, showing no sign of chaotic behaviour for this set of ICs.

4. Conclusions and further work

For two significant applications to do with DAEs, we have shown that differentiation of expressions, commonly done symbolically with the help of a computer algebra system, can be done efficiently and simply by AD.

First, for the Dummy Derivatives index-reduction method a theoretical scheme is given that applies in principle to preparing a DAE for solution by any standard DAE or ODE initial value code. DD-switching, which moves from one mode (local coordinate system) to another, is at the housekeeping level just a change of the size-DOF set S of indices (j, l) that define the state vector.

The scheme reduces order and index together, so one need not pre-reduce to first order form. Finally, following this paper's theme, differentiating DAE components f_i selectively (some more than others) does not need symbolic algebra; it can be done by standard AD methods of treating them as truncated power series.

We have proof-of-concept implementations in MATLAB and C++. It remains to be seen whether the scheme can be made efficient as a practical tool. For DAEs from industrial applications that may need to switch among very many modes, it may (as in the more general case of hybrid systems) be worth keeping a run time data base of modes used, if this can speed up re-entry to a mode that has been met before.

Second, we have shown that for a DAE, all or part of whose equations $f_i = 0$ derive from the Lagrangian L of a mechanical system, producing the f_i from L can be done by pure AD without symbolic algebra. The theory was illustrated by simulation examples: a constrained mechanical system, a forced pendulum as a prescribed-path control problem, and an ODE of planetary motion.

The method of directly solving from a Lagrangian by overlaying one AD type on another might be used with other DAE solvers and AD tools. However our infrastructure, of DAETS with FADBAD++ and our Lagrangian facility has several advantages:

- For any SA-friendly DAE, the user does not need to perform index/order reduction, since it is handled by DAETS automatically.

- It avoids large symbolic expressions that a computer algebra system typically generates when converting to a form suitable for integration by a standard ODE/DAE solver.
- It can be programmed in a way that is intuitive and close to the mathematics.
- It gives remarkably fast code in the cases we have tried.
- Constrained “first kind” Lagrangians in cartesian coordinates are often simpler to formulate than unconstrained “second kind” ones in generalised coordinates. For a high-index DAE solver such as DAETS, possible obstacles posed by index reduction and DD-switching are absent, and constrained systems are as easy to solve as unconstrained, which may make “first kind” forms more attractive.

Future work will explore our Lagrangian approach on a variety of research and engineering problems, and in particular rigid-body mechanics simulations and control problems. We are particularly interested in hybrid systems, because of their importance in industrial engineering.

Acknowledgments. We acknowledge with thanks the support given to JDP by the Leverhulme Trust and the Engineering and Physical Sciences Research Council (EPSRC), both of the UK; and NN, GT, and XL by the Canadian Natural Sciences and Engineering Research Council (NSERC).

References

- [1] K.E. Brenan, S.L. Campbell, and L.R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, 2nd ed., SIAM, Philadelphia, 1996.
- [2] W.H. Enright and J.D. Pryce, *Two FORTRAN packages for assessing initial value methods*, ACM Trans. Math. Softw. 13 (1987), pp. 1–27, Available at <http://doi.acm.org/10.1145/23002.27645>.
- [3] T.A. Germer, *vec3d.h, a C++ class for 3D vectors, Version 7.00* (2015), Available at pml.nist.gov/Scatmech/code/vector3d.h, accessed March 2017.
- [4] A. Griewank, D. Juedes, and J. Utke, *ADOL-C, a package for the automatic differentiation of algorithms written in C/C++*, ACM Trans. Math. Software 22 (1996), pp. 131–167.
- [5] W.B. Hayes, *Surfing on the edge: chaos versus near-integrability in the system of Jovian planets*, Monthly Notes Royal Astronomical Society 386 (2008), pp. 295–306.
- [6] A. Hindmarsh, P. Brown, K. Grant, S. Lee, R. Serban, D. Shumaker, and C. Woodward, *SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers*, ACM TOMS 31 (2005), pp. 363–396.
- [7] S.E. Mattsson and G. Söderlind, *Index reduction in differential-algebraic equations using dummy derivatives*, SIAM J. Sci. Comput. 14 (1993), pp. 677–692.
- [8] U. Naumann, *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*, no. 24 in Software, Environments, and Tools, SIAM, Philadelphia, PA, 2012, Available at <http://bookstore.siam.org/se24>.
- [9] N. Nedialkov and J. Pryce, *DAETS user guide*, Tech. Rep. CAS 08-08-NN, Department of Computing and Software, McMaster University, Hamilton, ON, Canada, 2013, 68 pages, DAETS is available at <http://www.cas.mcmaster.ca/~nedialk/daets>.
- [10] N.S. Nedialkov and J.D. Pryce, *Solving differential-algebraic equations by Taylor series (III): the DAETS code*, JNAIAM J. Numer. Anal. Indust. Appl. Math 3 (2008), pp. 61–80.
- [11] C.C. Pantelides, *The consistent initialization of differential-algebraic systems*, SIAM J. Sci. Stat. Comput. 9 (1988), pp. 213–231.
- [12] J.D. Pryce, *A simple structural analysis method for DAEs*, BIT Numerical Mathematics 41 (2001), pp. 364–394.
- [13] J.D. Pryce, *A simple approach to Dummy Derivatives for DAEs*, Tech. Rep., Cardiff University, 2015, in preparation.
- [14] R. Riasa and C. Tischendorf, *Qualitative features of matrix pencils and DAEs arising in circuit dynamics*, Dynamical Systems 22 (2007), pp. 107–131.
- [15] L. Scholz and A. Steinbrecher, *A combined structural-algebraic approach for the regularization of coupled systems of DAEs*, Tech. Rep. 30, Reihe des Instituts für Mathematik Technische Universität Berlin, Berlin, Germany, 2013.
- [16] D.E. Schwarz and C. Tischendorf, *Structural analysis for electric circuits and consequences for MNA*, Tech. Rep., Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät II, Institut für Mathematik, 1998.
- [17] O. Stauning and C. Bendtsen, *FADBAD++ web page* (2003), <http://www.imm.dtu.dk/fadbad.html>.
- [18] G. Tan, *Symbolic-numeric methods for improving structural analysis of differential-algebraic equations* (2015), the 2015 AMMCS-CAIMS Congress, Wilfrid Laurier University, Waterloo, ON, Canada.
- [19] Y. Zhu, E. Westbrook, J. Inoue, A. Chapoutot, C. Salama, M. Peralta, T. Martin, W. Taha, M. O'Malley, R. Cartwright, A. Ames, and R. Bhattacharya, *Mathematical Equations As Executable Models of Mechanical Systems*, in *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS '10*, Stockholm, Sweden, Available at <http://doi.acm.org/10.1145/1795194.1795196>, ACM, New York, NY, USA, 2010, pp. 1–11.
- [20] J. Zonneveld, *Automatic Numerical Integration*, Mathematical Centre tracts, Mathematisch Centrum, Amsterdam, 1970, Available at <https://books.google.co.uk/books?id=ThkoAAAACAAJ>.

Appendix A. Extract from code for planetary problem

This is the `fcn()` code for 3D motion of $n + 1$ gravitating bodies, where body 0 is the Sun and the x, y, z positions of the other bodies are relative to it. It was specialised to the problem in Example 3.3 by providing suitable input to the main program, not shown. Note this function is not restricted to 5 bodies: their number and masses are passed as parameters.

```

1  template <typename T>
2  void fcn( T t, const T *z, T *f, void *pp ) {
3      const double *param = (double *)pp;
4      const int nMASS = param[0],
5              n = nMASS-1;
6      const double G = param[1];
7      const double *m = param + 2,
8              *mplanet = m+1;           // the masses EXCLUDING the Sun
9      const double Mtotal = (m + nMASS)[0]; // total mass, calculated in main program
10
11      typedef Vector3D< B<T> > vec3;
12      vector< B<T> > q(3*n), qp(3*n); // independent variables
13      B<T> L; // for storing Lagrangian
14      // { ... } ensures all intermediate variables go out of scope
15      {
16          init_q_qp(z,q,qp); // setup q, qp
17
18          // Convert to a vector of 3D vectors.
19          vector< vec3 > Q(n), Qp(n);
20          for (int imass=0; imass<n; imass++) {
21              Q[imass] = vec3( q [3*imass], q[3*imass+1], q [3*imass+2] );
22              Qp[imass] = vec3( qp[3*imass], qp[3*imass+1], qp[3*imass+2] );
23          }
24
25          vector< vec3 > R(nMASS), Rp(nMASS);
26          R[0] = Rp[0] = vec3(0,0,0);
27          for (int imass=0; imass<n; imass++) {
28              R [0] -= mplanet[imass]*Q [imass]; Rp[0] -= mplanet[imass]*Qp[imass];
29          }
30          R [0] /= Mtotal; Rp[0] /= Mtotal;
31
32          // then set r_1, ..., r_n and their derivatives:
33          for (int imass=1; imass<nMASS; imass++) {
34              R [imass] = Q [imass-1]+R [0]; Rp[imass] = Qp[imass-1]+Rp[0];
35          }
36
37          // Compute KE and PE in terms of r and rp arrays
38          B<T> KE = 0;
39          for (int imass=0; imass<nMASS; imass++)
40              KE += m[imass] * Rp[imass]*Rp[imass];
41          KE *= 0.5;
42
43          // Potential Energy (sum of all mass-to-mass PEs, -> -oo as bodies
44          // become close)
45          B<T> PE = 0;
46          for (int imass=0; imass<nMASS; imass++)
47              for (int jmass=imass+1; jmass<nMASS; jmass++)
48                  PE -= m[imass] * m[jmass] / norm(R[imass]-R[jmass]);
49          PE = G*PE; // bring in gravitational constant
50
51          L = KE-PE;
52      }
53      vector<B<T> > C; // dummy constraint variable
54      setupEquations(L,z,q,qp,C,f);
55  }

```